Flash Memory Management for Lightweight Storage Systems[1]

Mei-Ling Chiang[†, ‡] *Paul C. H. Lee*[‡] *Ruei-Chuan Chang*[†, ‡]

Institute of Information Science[‡]
Academia Sinica, Nan Kang, Taipei, ROC

Department of Computer and Information Science[†]
National Chiao Tung University,
Hsinchu, Taiwan, ROC

ABSTRACT

This report describes a management scheme for flash memory-based storage systems. This scheme will be implemented into a flash memory-based FAT file system in the *Ramos* project running in the Institute of Information Science (IIS). *Ramos* project targets on building software components for real time and multimedia applications that most of the consumer electronic products can be easily constructed by these software components. The flash memory-based file system is especially important to those products because flash memory is lightweight, shock-resistant, nonvolatile, power-efficient and is wildly used in these products.

Flash memory has many advantages. However, the specific erase-operations before writing into flash memory are slow and power-wasted, which usually decrease system performance and consume the limited battery power. In addition to that, the number of flash memory erase cycles is also limited. In order to reduce the number of erase operations needed and to evenly wear flash memory, a new flash memory management scheme is proposed in this report. A new cleaning policy is also introduced to reduce the number of erase operations. Performance evaluations show that erase operations can be reduced by 55%.

[1] This work is a part of *Ramos* project for improving flash-memory based storage system used in consumer electronic products.

1. INTRODUCTION

Flash memory is small, lightweight, shock resistant, nonvolatile, and consumes less power than traditional storage media [6,7,11]. These features show promises for use flash memory as storage devices in consumer electronics, embedded systems, and mobile computers. Examples are digital cameras, set-top boxes, cellular phones, audio recorder, notebooks, hand-held computers, and Personal Digital Assistants (PDAs) [2,8].

Flash memory must be used with care because of the special hardware limitations [6,11,12] shown in Table 1. Flash memory is partitioned into *segments*[2] defined by hardware manufactures, and these segments cannot be overwritten unless first erased. The erase operation is performed only on full segments, which is slow and power-wasted. Segments also have limited number of program/erase cycles (e.g., 1,000,000 times for the Intel Series 2+ Flash Memory Cards [11,12]). Therefore, erase operations should be avoided for better performance and longer flash memory lifetimes. To avoid wearing specific segments out which would affect the usefulness of the whole flash memory, data should be written evenly to all segments. This is called *even wearing* or *wear-leveling* [6].

Read Cycle Time	150 ~ 250 ns
Write Cycle Time	6 ~ 9 *u*s/byte
Block Erase Time	0.6 ~ 0.8 sec
Erase Block Size	64 or 128 Kbytes
Erase Cycles Per Block	1,000,000

Table 1: Flash memory characteristics

Since segments are relatively large (e.g., 64 Kbytes or 128 Kbytes for Intel Series 2+ Flash Memory Cards), updating data in place is not efficient because all segment data must first be copied to a system buffer, and then updated. Then, after the segment has been erased, all data must be written back from system buffer to the segment. Thus, updating even one byte data requires one slow erase and several write operations. Besides, flash memory blocks of hot spots would soon be worn out.

[2] We use “segment” to represent hardware-defined erase block, and “block” to represent software-defined block.

To avoid having to erase during every update, we design a **F**lash **M**emory **S**erver (FMS) that uses a *non-in-place-update* scheme to manage data in flash memory. That is, instead of updating data at the same address, updates are written to any empty space in flash memory and the obsolete data are left at the same address as garbage, which a software cleaning process will reclaim. Thus, data update is efficient when cleaning operations can be performed in the background.

There exist many *cleaning policies*, which control cleaning operations such as, when to clean, which segment to clean, and where to write update data. The *greedy* policy [13,15,16,19] always chooses least utilized segments for cleaning, and *cost-benefit* policy [13,15,16] chooses those segments that maximize the formula: $\frac{a*(1-u)}{2u}$, where u is the percentage of valid data in the segment and a is the times since the most recent modification. This a is used as an estimate of how long the space is likely to stay free.

In this report, we propose a new cleaning policy: the **C**ost **A**ge **T**imes policy (CAT), which selects segments for cleaning according to cleaning cost, ages of data in segments, and the number of times the segment has been erased. Data blocks are classified into three types: read-only, hot, and cold, and are clustered separately when cleaning. This fine-grained separation of different types of data can reduce flash memory cleaning costs. An even wearing method is also proposed.

The FMS server allows applications to select various cleaning policies such as greedy, cost-benefit, and CAT policies. Performance evaluations show that CAT policy significantly reduces the number of erase operations performed and the cleaning overhead, but also evenly wear flash memory. CAT policy performed best and incurred 55% fewer erase operations than greedy policy and 29% fewer than cost-benefit policy under high locality of references

The rest of this report is organized as follows. Section 2 describes related work. Section 3 describes the design and implementation of Flash Memory Server. Section 4 presents our cleaning policies. Section 5 presents performance evaluation results, and Section 6 concludes this report.

2. RELATED WORK

Several storage systems and file systems have been developed for flash memories. *SanDisk* uses flash memory as a disk emulator that supports the DOS FAT system. Segment sizes are small and equal to disk block sizes (512 bytes). *In-place-update* is used, and segments are erased before updating. However, erase operations are slow. In-place-update must be accompanied by asynchronous cleaning to improve write performance [7].

Some flash memory products, such as Intel Series 2+ flash memory [11,12], have large segment sizes (64 Kbytes or 128 Kbytes), so the *non-in-place-update* approach is generally used. Wu and Zwaenepoel [19] proposed a storage system for flash memory, the eNVy, which uses copy-on-write and page-re-mapping techniques to avoid updating data in place. The proposed *hybrid cleaning* method combining FIFO and Locality-gathering in cleaning segments aims to perform well for uniform accessing and highly localized of reference.

Rosenblum et al. [15] suggested that the Log-Structured File System [15,16,17], which writes data as append-only logs instead in-place update can be applied to flash memories. Kawaguchi et al. [13] used a log approach similar to LFS to design a flash-memory-based file system for UNIX. *Separate segment cleaning*, which separates cold segments and hot segments, was also proposed.

Microsoft's Flash File System (MFFS) uses a linked-list structure and supports the DOS FAT system [18]. The greedy method is used to clean segments. David Hinds [9,10] implemented flash memory drivers in the Linux PCMCIA [1] package that uses the greedy method most times, but can choose to clean segments erased fewest times to ensure even wear.

3. SYSTEM DESIGN AND IMPLEMENTATION

We use the *non-in-place-update* scheme in our FMS server to manage data in flash memory to avoid having to erase during every update. That is, instead of updating data in place, data updates are written to any empty space, and obsolete data are left as garbage. When the number of free segments is below a certain threshold, a software cleaning process, *cleaner*, begins to reclaim garbage. Every data block is associated with a unique constant logical block number. When data blocks are updated, their physical positions in flash memory change.

We first describe data structure on flash memory in Section 3.1. We then describe the non-in-place-update scheme in detail and compare it with in-place-update scheme in Section 3.2. Two tables, the translation table and the lookup table, are maintained in main memory to speed up processing. These tables are described in Section 3.3 and Section 3.4, respectively. FMS server uses separate segments to manage different types of data. Section 3.5 describes the separate segment management.

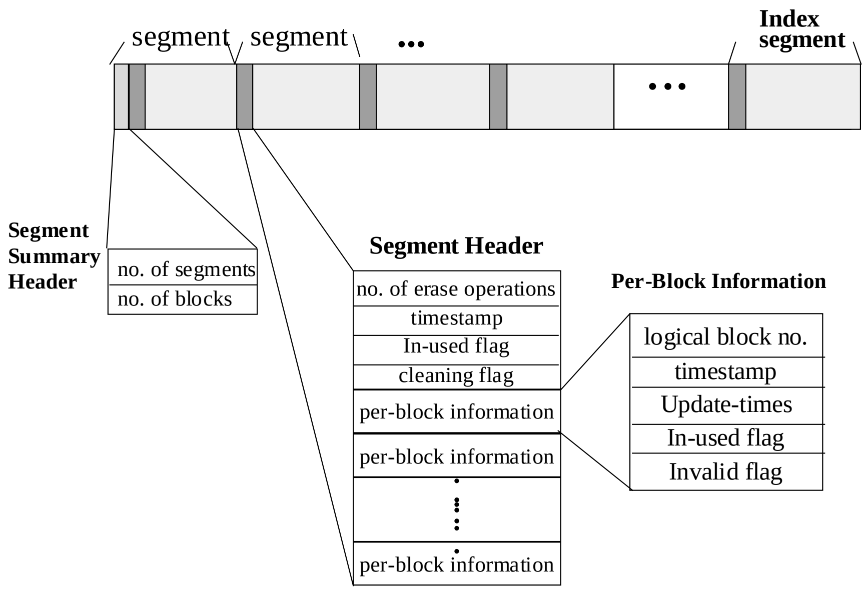


Figure 1: Data structures on flash memory

3.1 Data Structures on Flash Memory

FMS server manages flash memory as fixed-size blocks. The data structures on flash memory are shown in Figure 1. Each segment has a *segment header* to record segment information, such as the number of times a segment has been erased, a timestamp, control flags for cleaning, and the *per-block information array*. The per-block information array contains information about each block in the segment, such as the corresponding logical block number, the timestamp, the number of times that a block been updated, and invalid flag. The invalid flag indicates whether a block is obsolete. The *segment summary header*, located in the first segment, records global information about flash memory, such as the total number of segments in flash memory and the total number of blocks in each segment.

3.2 Non-In-Place-Update Operations

In-place-update requires one erase and several write operations as shown in Figure 2. It is especially useful when erase segment size is small and equal to the block size which storage systems maintain. However, in-place-update is inefficient in systems that have large erase segment sizes.

```
Write()
{
    If new write {
        Allocate a free block;
        Write out data into the free block ;
    } else
        In-place-update();
}

In-place-update()
{
      /* update a block in place */
      Read all data in the segment into a system buffer;
      Update data in the system buffer;
      Erase the segment;
      Write back all data from system buffer to segment;
}
```

Figure 2: In-place-update operations

```
Write()
{
    If new write {
        Allocate a free block;
        Write out data into the free block;
    } else
        Non-in-place-update();
}

Non-in-place-update()
{     /* not to update a block in place */
      Mark the obsolete block as invalid in per-block information;
      Allocate a free block;
      Write out data into the free block;
}

Cleaning()
{     /* when system runs out of free space */
      Select a victim segment for cleaning;
      Identify valid data in the victim segment from segment header;
      Copy out valid data to another clean flash memory space;
      Erase the victim segment
}
```

Figure 3: Non-in-place-update and cleaning operations

Instead of updating data in place, we use *non-in-place-update* scheme. That is, before a data block is modified, the original copy in flash memory is marked invalid as obsolete in per-block information. Then updates are written to another newly allocated free block. Figure 3 shows the detailed operations.

3.3 Block-based Translation Table

Since data blocks are not updated in place, their physical locations in flash memory change when updated. The *translation table* is constructed to record the physical location for each logical block to speed up the address translation from logical block numbers to physical addresses in flash memory. Therefore, every logical block has an entry in the translation table to record segment number and block number. Figure 4 shows the translation table and address translation.

Initially, the translation table is constructed in main memory during FMS server startup time by reading all segment headers from flash memory. When data blocks are updated to another new empty blocks, the corresponding table entries are updated to record current physical locations.

3.4 Segment-based Lookup Table

The *lookup table*, shown in Figure 5, is constructed in main memory to record information about each segment, such as number of times the segment has been erased, segment creation time, and control flags for cleaning. Initially, this segment-information is obtained by reading all segment headers from flash memory during FMS server startup time. During run time, FMS does bookkeeping of *valid block count* to count the number of valid blocks in a segment. Cleaner then uses these segment information to speed up the process of selecting segments for cleaning. Since the table is constructed in main memory, the selection process is fast. The table also contains the *first free block* to indicate the first free block available for writing in a segment. It is used to speed up block allocation.

3.5 Separate Segments Management

We classify data blocks as *read-only*, *hot*, and *cold* blocks. Read-only data once created will never be modified, and are allocated in segments dedicated for them. Data blocks are defined as hot when updated frequently at recent time, otherwise they are defined as cold. During cleaning, hot valid blocks in cleaned segments are distributed to hot segments, while cold valid blocks are distributed to cold segments. Therefore, three segment lists are used: the read-only segment list, the hot segment list, and the cold

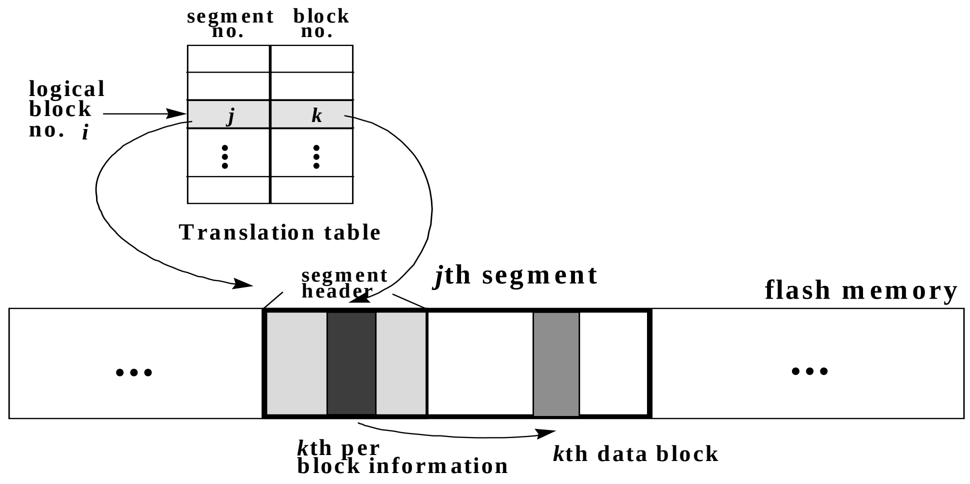


Figure 4: Translation table and address translation

	Erase count	Timestamp	Used flag	Cleaning flag	Valid blocks count	First free block no.
	.	.	.	.	.	.
Segment no. *i*						
	.	.	.	.	.	.

Figure 5: Lookup table to speed up cleaning

segment list, as shown in Figure 6. The *active- segments index* records segments that are currently used for writing data in segment lists. When changed, it is appended as log in the final segment. When final segment is out of free space, the final segment is erased first before wrapping around the log.

The *free-segment list* records information about available free segments. Initially, FMS server reads segment headers from flash memory to identify free segments to construct the free-segment list at server startup time.

4. CLEANING POLICIES

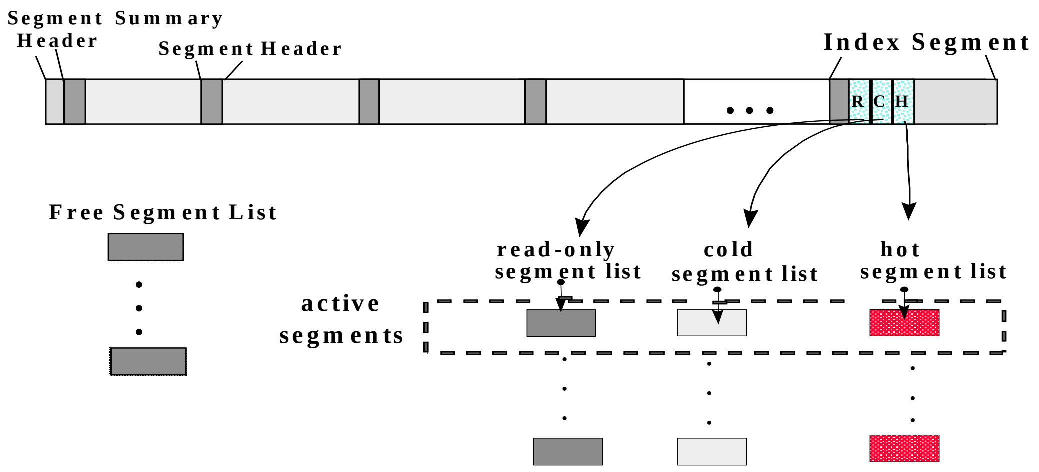


Figure 6: Three types of segment lists, and free-segment list

Cleaning policies control cleaning operations, such as when to clean, which segments to clean, and where to write the update data. When cleaning, valid blocks in cleaned segments are distributed to separate segments depending on whether valid blocks are cold or not. The idea is to cluster data blocks separately according to their types such that segments are full of all hot data or all cold data. Because hot data are updated frequently and soon become garbage, segments containing all hot data would soon come to contain the largest amounts of invalidated spaces. Cleaning these segments can reclaim the largest amounts of space, therefore, flash memory cleaning cost can be significantly reduced.

The cleaner selects segments that minimize the following formula for cleaning.

$$\boldsymbol{CleaningCost_{FlashMemory}} * \frac{1}{Age} * \boldsymbol{CleaningTimes},$$

This formula is called the ***C****ost* ***A****ge* ***T****imes* (CAT) formula. The *cleaning cost* is defined as the cost of every useful write to flash memory as $\frac{u}{1-u}$. The *u* is the percentage of valid data in the segment to be cleaned. Every (*1-u*) write entails the cleaning cost of writing out *u* valid data. *Age* is defined as the elapsed time since the segment was created. *Cleaning times* counts the numbers of erase operations conducted on segments. The basic idea of CAT formula is to minimize segment-cleaning costs, but give segments just cleaned more time to accumulate garbage for reclamation. In addition, segments erased fewest times are given more chances to be selected for cleaning. This avoids concentrating cleaning activities on a few segments, thus allowing more even wearing.

To avoid wearing specific segments out and thus limiting the usefulness of the whole flash memory, we swap the segment erased most times and the segment erased fewest times when a segment is reaching its projected lifecycle limit.

5. EXPERIMENTAL RESULTS

We have implemented the FMS server on Linux Slackware96 in GNU C++. We used a 24 Mbyte Intel Series 2+ Flash Memory Card [11,12]. Table 2 summarizes the experimental environment. To measure the effectiveness of alternate cleaning policies, a variety of cleaning policies were implemented in the FMS server. Since cleaning activity is affected by data-access patterns, workloads with various data-accessing patterns are measured, such as sequential accesses, random accesses, and locality of references. We focused on data updates that incurred invalidation of old blocks and writing of new blocks.

Three policies were measured: *Greedy* represents the greedy policy [13,15,16,19] with no

Hardware: PC: Intel 486 DX33, 32 Mbytes of RAM PC Card Interface Controller: Intel PCIC Vadem VG-468 Flash memory: Intel Series 2+ 24Mbyte Flash Memory Card (segment size:128 Kbytes) HD: Seagate ST31230N 1.0 G
Software: Operating system: Linux Slackware 96 (Kernel version: 2.0.0, PCMCIA package version: 2.9.5)

Table 2: Experimental environment

separations of hot and cold blocks. *Cost-benefit* represents the cost-benefit policy [13] with separate segment cleaning for hot and cold segments. *CAT* represents our CAT policy with fine-grained separation of hot and cold blocks. The segment selection algorithms and data redistribution methods are different for these policies.

Initially, we wrote enough blocks in sequence to fill the flash memory to the desired level of utilization. Benchmarks were created to update the initial data according to the required access patterns. Total 192 Mbytes of data were written to the flash memory in 4 Kbyte units. The FMS server manages data in 4 Kbyte blocks. All measurements were made on a freshly start of the system and averaging four runs. For each measurement, the numbers of erase operations conducted on segments and the numbers of blocks copied during cleaning were counted to measure the effectiveness of each policy.

We found that our policy significantly reduced the number of erase operations performed on segments and the number of blocks copied during cleaning, as described in Section 5.1. Section 5.2 shows that as the locality of reference increased, the advantage of our policy over other policies increased dramatically. Even wearing of flash memory is described in Section 5.3. Section 5.4 shows that as flash memory utilization increased, our policy outperformed others by a large margin.

5.1 Cleaning Effectiveness of Various Cleaning Policies

Table 3 shows the results. Each policy performed equally well for sequential access. No blocks were copied since sequential updating caused invalidation of each block in the cleaned segment. Greedy performed best for random access. Cost-benefit and CAT performed similarly. CAT incurred 2.4% more erase operations than Greedy did.

Under high locality of reference that 90% of the write accesses went to 10% of the initial data, CAT performed best. CAT incurred 55% less erase operations than Greedy did, and 29% less than Cost-benefit. The number of blocks copied during cleaning for CAT was 66.8% less than Greedy and 40.17% less than Cost-benefit.

Since large amount of erased operations and blocks copied were reduced in CAT, the average throughput of CAT was much better than Greedy and Cost-benefit, as shown in Figure 7. The average throughput of CAT was 99.73% better than Greedy and 52.58% better than Cost-benefit.

	Greedy		Cost-Benefit		CAT	
	Number of erased operation	Number of blocks copied	Number of erased operation	Number of blocks copied	Number of erased operation	Number of blocks copied
Sequential	1567	0	1568	0	1568	0
Random	7103	171624	7274	176913	7276	176542
Locality	8827	225068	5596	124888	3978	74726

Initial data 90% (20.5 MB), Total data written: 192 MB

Table 3: Performance of various cleaning policies

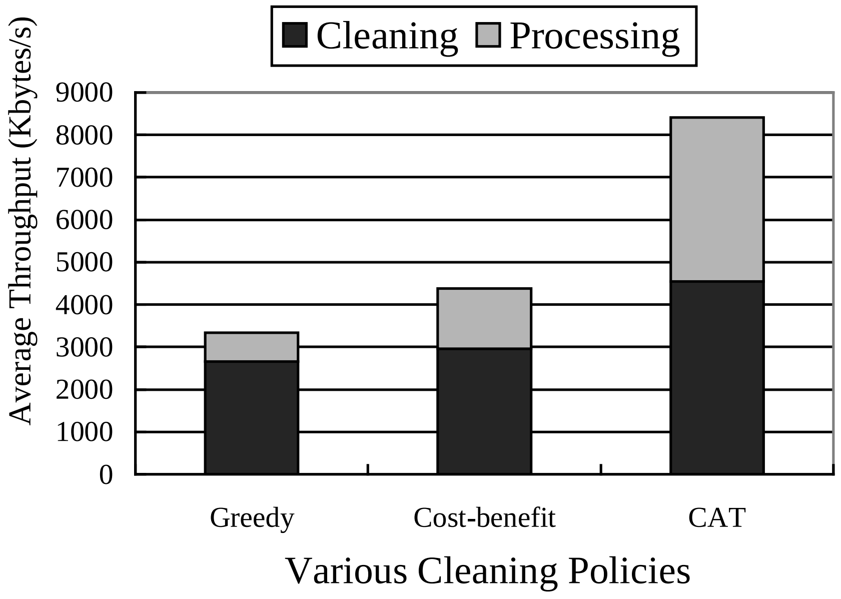
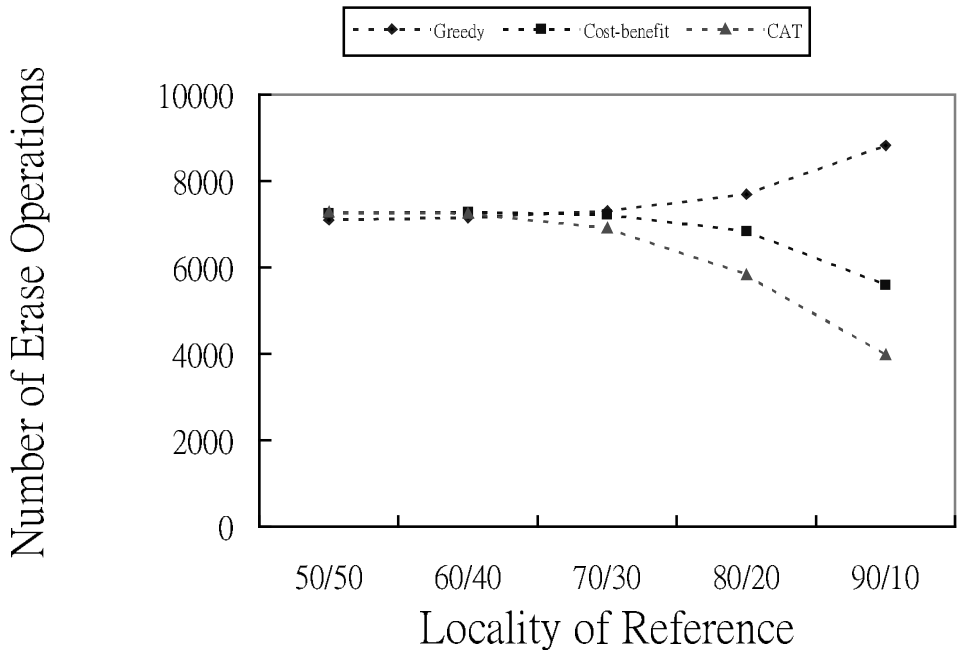
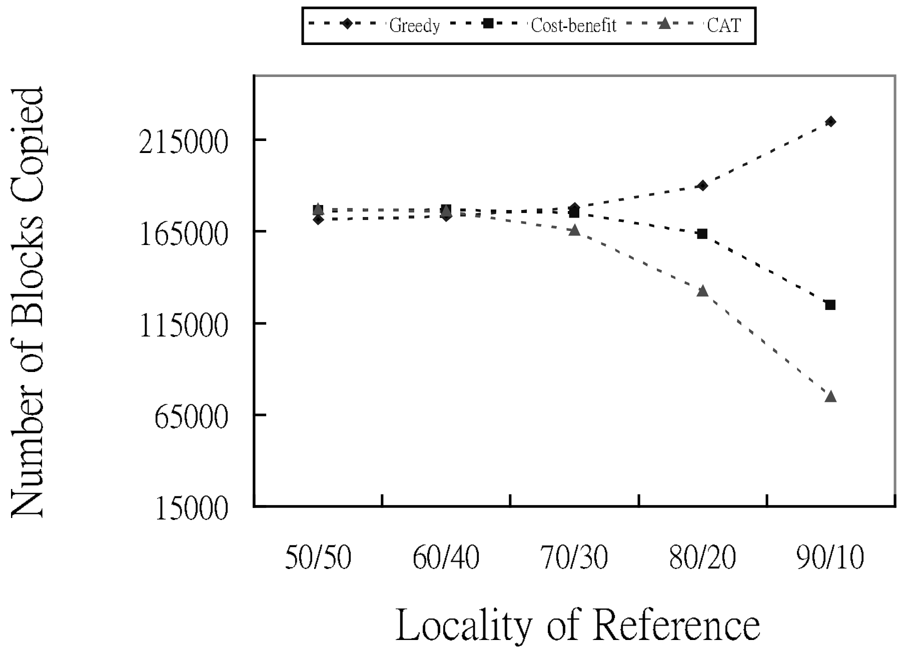


Figure 7: Average throughput

To sum up, no single policy performed well for all data access patterns. Performance differences between CAT and other policies were small under sequential and random accesses. However, CAT significantly outperformed the other policies for high locality of references. Though CAT required more processing time to cluster data during cleaning time, however, CAT eliminated significant numbers of erase operations and blocks copied. Therefore throughput were largely improved. It is beneficial to use fine-grained method for data clustering.



(a) Effects on erase operations



(b) Effect on the number of blocks copied during cleaning

Figure 8: Varying the locality of reference

5.2 Effect of Locality of Referencing

Figure 8 shows how locality of reference performance varied among policies. We use the notation for locality of reference as “x/y” that x% of all accesses go to y% of the data while (1-x)% of accesses to go to the remaining (1-y)% of data. CAT outperformed the other policies when 70% of the accesses were to 30% of the data. As the locality of reference was increased, the performance of CAT increased rapidly and the performance of Greedy deteriorated severely. The performance advantage of CAT over Greedy and Cost-benefit increased dramatically as well. This is because CAT uses fine-grained methods to separately gather hot and cold data, such that cold data are less likely to mix with hot data. Therefore, the number of erase operations performed and the number of blocks copied during cleaning are significantly reduced.

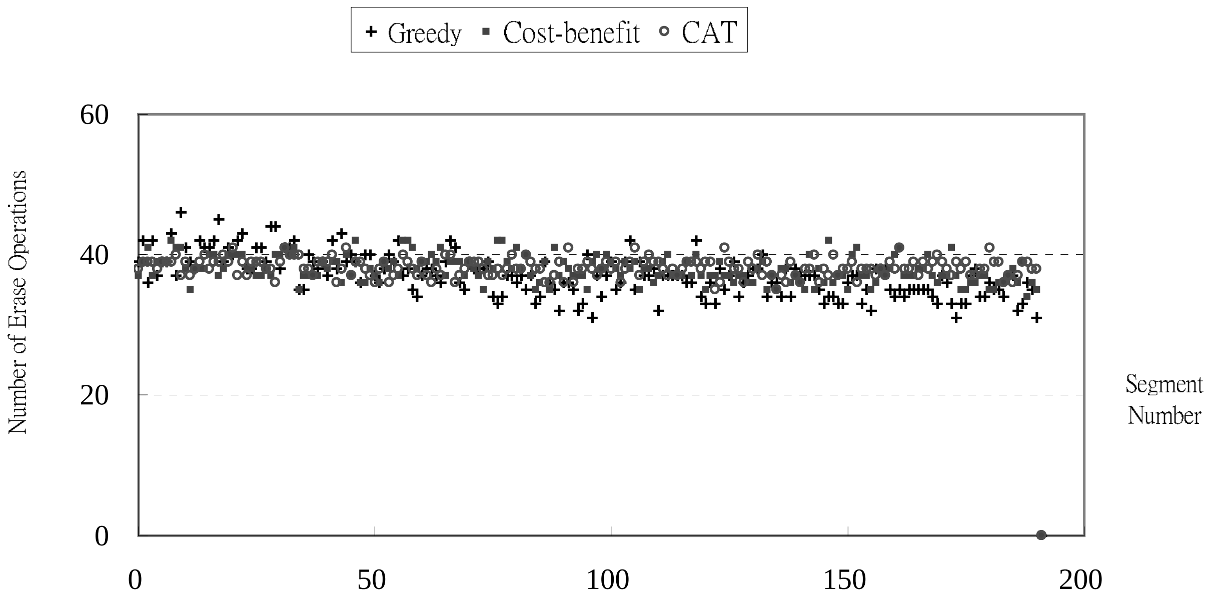
5.3 Effect of Even Wearing

To explore each policy’s degree of Wear-Leveling, we created a utility to read the number of erase operations performed on each segment in flash memory. Then the standard deviation of these numbers was calculated to represent the degree of Wear-Leveling. The smaller the deviation, the more evenly the flash memory is worn.

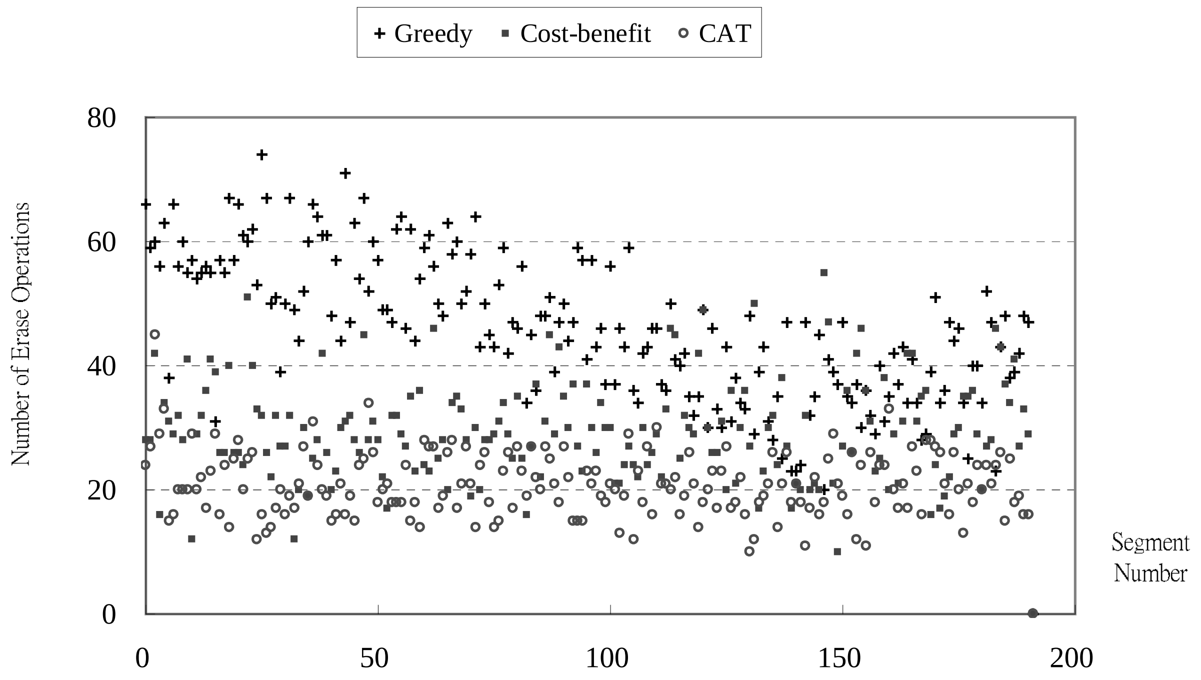
Because all policies incurred similar erase operation distributions under sequential accessing, we show only the cleaning distributions under random accessing and locality accessing in Figure 9. Though, under random accessing, CAT incurred slightly more erase operations than the other policies, it performed best in degree of wear-leveling, as shown in Figure 9(a). The standard deviation was 3.04 for CAT, 3.98 for Greedy, and 3.3 for Cost-benefit.

Figure 9(b) shows the significant differences among these policies under high locality of reference, in which 90% of the accesses were to 10% of the data. CAT incurred many fewer erase operations than the other policies, and also exhibited little variation, while other policies varied widely. The standard deviation was 5.38 for CAT, 11.85 for Greedy, and 8.3 for Cost-benefit. This is because only CAT considers about evenly wearing when choosing segments to clean. The segments erased fewest times are given more chances to be selected to clean in CAT.

5.4 Impact of Flash Memory Utilization



(a) Random access



(b) High locality of reference (90% of accesses to 10% of data)

Figure 9: Cleaning distributions for various cleaning policies

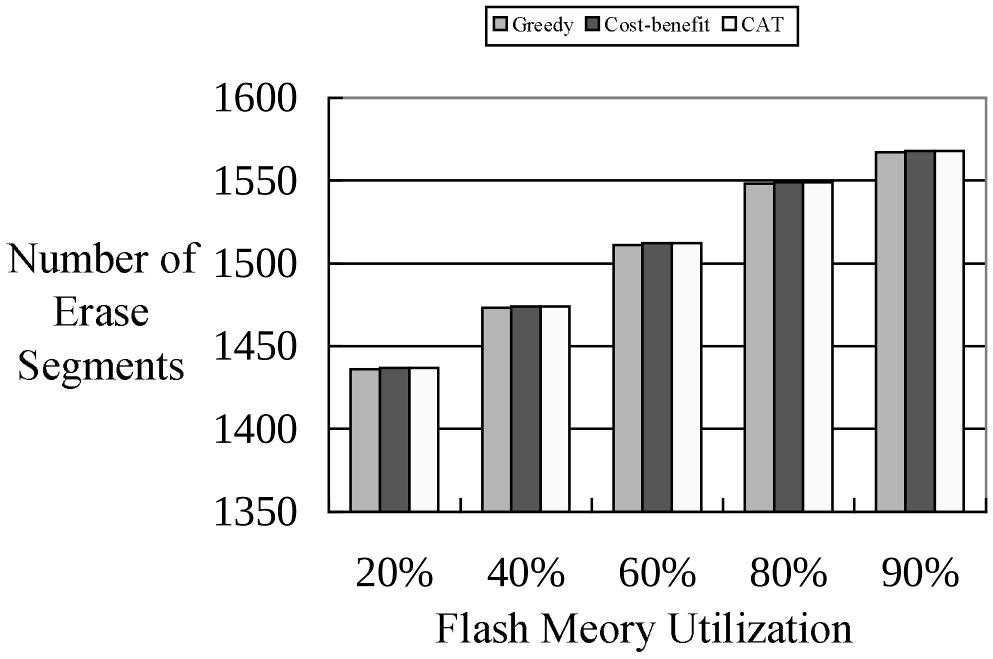
We want to investigate the impact of flash memory utilization on the cleaning overhead. Initially, the flash memory was filled with various percentages of data. Then 192 Mbytes of data were overwritten to the initial data in 4 Kbyte units. Figure 10 shows the results of varying the flash memory utilization. As shown in Figure 10(a)(b), all policies performed similarly for sequential and random accesses. Performances decreased as utilization increased since less free space was left and more cleaning had to be performed. However, under high locality of reference, as utilization increased, Greedy degraded dramatically while CAT degraded much more gracefully, as shown in Figure 10(c). Besides, the performance advantages of CAT over other policies increased greatly too.

6. CONCLUSIONS

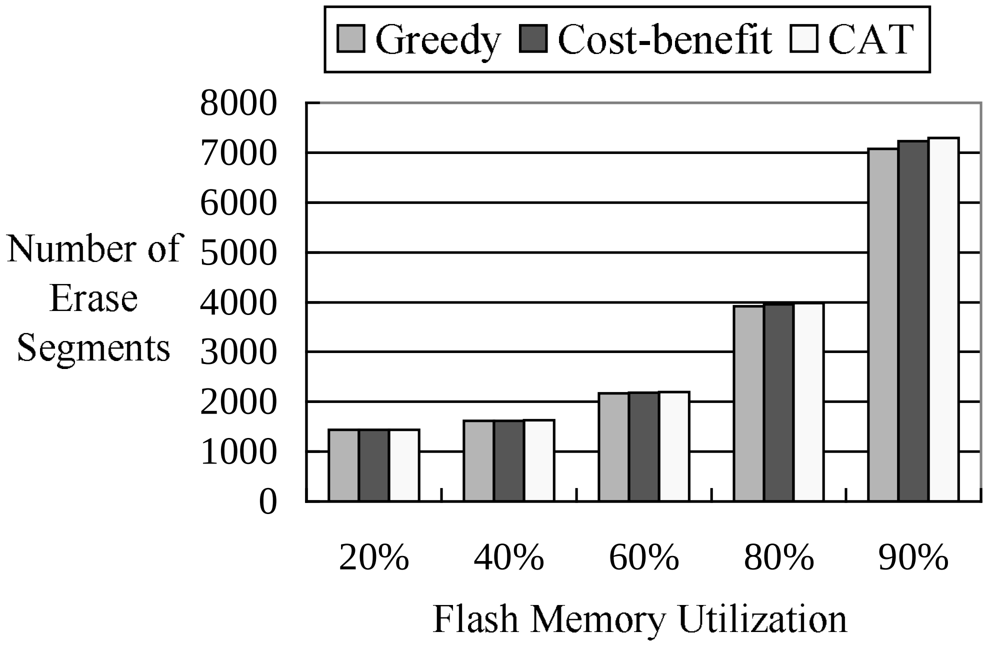
In this report we describe the design and implementation of FMS, a storage server utilizing flash memory. The server uses non-in-place-update approach to avoid having to erase during every update, and employs a fine-grained method to cluster hot, cold and read-only data into separate segments. A new cleaning policy, the CAT policy, is also proposed to reduce the number of erase operations performed and to evenly wear flash memory. The CAT policy selects segments for cleaning according to utilization, age of the data, and the number of erase operations performed on segments.

Performance evaluations show that with this CAT policy and fine-grained separation of hot and cold data, the proposed storage server not only significantly reduces the number of erase operations required, but also evenly wear out flash memory. Therefore, flash memory lifetime is extended and cleaning overhead is reduced. The number of blocks copied during cleaning are significantly reduced as well.

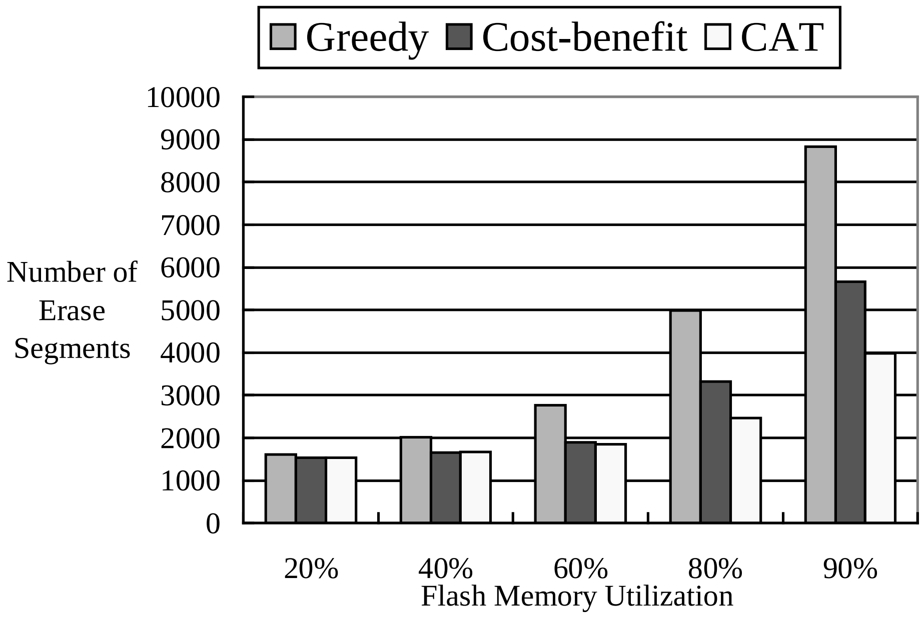
Future work can be summarized as follows. First, the FMS will be implemented on *Ramos* kernel to serve real applications' need. We will do performance tuning of the FMS server after the implementation is finished. Next, we will use extensive numbers of real applications or traces to examine the effectiveness of the proposed cleaning policies on our FMS server. Last, we will integrate the FMS server into ROSS [5], a RAM-based Object Storage Server designed for PDAs, to enable ROSS to store data in external flash storage. ROSS will also be implemented on *Ramos* to compare object-based storage system with FAT file system.



(a) Sequential access



(b) Random access



(c) Locality access

Figure 10: Performance under various flash memory utilizations

REFERENCES

[1] D. Anderson, *PCMCIA System Architecture*, MindShare, Inc. Addison-Wesley Publishing Company, 1995.

[2] N. Ballard, "State of PDAs and Other Pen-Based Systems," In *Pen Computing Magazine*, Aug. 1994, pp. 14-19.

[3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-Volatile Memory for Fast, Reliable File Systems," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[4] R. Caceres, F. Douglis, K. Li, and B. Marsh, "Operating System Implications of Solid-State Mobile Computers," *Fourth Workshop on Workstation Operating Systems*, Oct. 1993.

[5] M. L. Chiang, S. Y. Lo, Paul C. H. Lee, and R. C. Chang, “Design and Implementation of a Memory-Based Object Server for Hand-held Computers,” *Journal of Information Science and Engineering,* vol. 13, 1997.

[6] B. Dipert and M. Levy, *Designing with Flash Memory,* Annabooks, 1993.

[7] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage Alternatives for Mobile Computers," *Proceedings of the 1st Symposium on Operating Systems Design and Implementation,* 1994.

[8] T. R. Halfhill, "PDAs Arrive But Aren't Quite Here Yet," *BYTE,* Vol. 18, No. 11, 1993, pp. 66-86.

[9] D. Hinds, “Linux PCMCIA HOWTO,” http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-HOWTO.html.

[10] D. Hinds, “Linux PCMCIA Programmer’s Guide,” http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-PROG.html.

[11] Intel, *Flash Memory, 1994.*

[12] Intel Corp., “Series 2+ Flash Memory Card Family Datasheet,” http://www.intel.com/design/flcard/datashts, 1997.

[13] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proceedings of the 1995 USENIX Technical Conference*, Jan. 1995.

[14] B. Marsh, F. Douglis, and P. Krishnan, "Flash Memory File Caching for Mobile Computers," *Proceedings of the 27 Hawaii International Conference on System Sciences,* 1994.

[15] M. Rosenblum, “The Design and Implementation of a Log-Structured File System,” *PhD Thesis*, University of California, Berkeley, Jun. 1992.

[16] M. Rosenblum and J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” *ACM Transactions on Computer Systems*, Vol. 10, No. 1, 1992.

[17] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, “An Implementation of a Log-Structured File System for UNIX,” *Proceedings of the 1993 Winter USENIX*, 1993.

[18] P. Torelli, "The Microsoft Flash File System," *Dr. Dobb’s Journal*, Feb. 1995.

[19] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.